Implementing autonomic administration DSLs in TUNe

Suzy Temate, Alain Tchana, Laurent Broto, Hagimont Daniel

Implementing autonomic administration DSLs in TUNe

S. Temate, A. Tchana, L. Broto, and D. Hagimont

IRIT-ENSEEIHT, 2 rue Camichel, BP 7122, 31071 Toulouse, France
lastname@irit.fr

Abstract. Software components are recognized as the most adequate approach to support autonomic administration systems. We implemented and experimented with such a system, but observed that the interfaces of a component model are too low-level and difficult to use. Consequently, we designed higher abstraction level languages for modeling administration policies. These languages are specific to our autonomic administration domain. We metamodeled and implemented these DSLs on the Kermeta framework.

1 Introduction

Autonomic computing is recognized as the most promizing approach to deal with the complexity of today's software environments. It aims at providing system support for deploying and configuring applications in a distributed environment, monitoring application's execution in that environment in order to detect incidents such as failures or overloads, and reconfiguring applications in response to such incidents.

Many works in this area have relied on a component model to provide such an autonomic system support [1–3]. The basic idea is to encapsulate the managed elements (legacy software) in software components (called wrappers) and to administrate the environment as a component architecture. Then, the administrator benefits from a uniform management interface (the interface of the component model the autonomic system relies on) whatever is the legacy software he has to administrate.

However, even if components bring many advantages, autonomic systems are still difficult to use. In most of the cases, defining an autonomic management policy requires to program it using the programming interfaces of the underlying component model, which are too low-level and difficult to use. This led us to explore the introduction of higher level formalisms for the specification of all the administration tasks (wrapping, configuration, deployment, reconfiguration).

In the software engineering community, such formalisms are called Domain Specific Languages (DSL), and the model driven engineering (MDE) approach introduces abstractions and tools for implementing such DSLs.

In this paper, we present the TUNe autonomic administration system that we designed and implemented. TUne introduces administration policy specification

DSLs which help administrators in defining their policies. Relying on the MDE approach, we defined the metamodels associated with these DSLs and used the Kermeta toolkit to implement the DSLs' runtime and editing tools.

The rest of the paper is structured as follows. Section 2 presents the context of this work and the issues which motivate this work. Section 3 describes TUNe's policy specification DSLs. The implementation of these DSLs on Kermeta is presented in Section 4. After a review of related works in Section 5, we conclude in Section 6.

2 Problem statement

In this section, we first present an application case that we use to illustrate our contributions. We then present in more details what we mean by component-based autonomic management.

2.1 J2EE Use Case

The Java 2 Platform, Enterprise Edition (J2EE) defines a model for developing web applications [4] in a multi-tiered architecture. Such applications are typically composed of a web server (e.g. Apache), an application server (e.g. Tomcat) and a database server (e.g. MySQL). Upon an HTTP client request, either the request targets a static web document, in which case the web server directly returns that document to the client; or the request refers to a dynamically generated document, in which case the web server forwards that request to the application server. When the application server receives a request, it runs one or more software components (e.g. Servlets, EJBs) that query a database through a JDBC driver (Java DataBase Connection driver). Finally, the resulting information is used to generate a web document that is returned to the web client.

In this context, the increasing number of Internet users has led to the need of highly scalable and highly available services. To face high loads and provide higher scalability of Internet services, a commonly used approach is the replication of servers in clusters. Such an approach (Figure 1) usually defines a particular software component in front of each set of replicated servers, which dynamically balances the load among the replicas. Here, different load balancing algorithms may be used, e.g. Random, Round-Robin, etc.

This example is characteristic of the management of a distributed software infrastructure where very heterogeneous servers are distributely deployed, configured and interconnected in order to provide a global service. The management of the whole infrastructure can be very complex and requires a lot of expertise. Many files have to be edited and configured consistently. Also, failures or load peaks (when the chosen degree of replication is too low) must be treated manually.

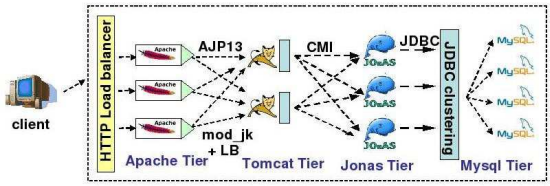

Fig. 1. Clustered J2EE servers

2.2 Component-Based Autonomic Computing

Component-based management aims at providing a uniform view of a software environment composed of different types of servers. Each managed server is encapsulated into a component and the software environment is abstracted as a component architecture. Therefore, deploying, configuring and reconfiguring the software environment is achieved by using the tools associated with the used component-based middleware.

The component model we used in TUNe is the Fractal component model [5]. Any software managed with TUNe is wrapped into a Fractal component which interfaces its administration procedures. Therefore, the Fractal component model is used to implement a management layer (Figure 2) on top of the legacy layer (composed of the actual managed software). In the management layer, all components provide a management interface for the encapsulated software, and the corresponding implementation (the wrapper) is specific to each software (e.g. the Apache web server in the case of J2EE). Fractal's control interfaces allow managing the element's attributes and bindings with other elements, and the management interface of each component allows controlling its internal configuration state. Relying on this management layer, sophisticated administration programs can be implemented, without having to deal with complex, proprietary configuration interfaces (generally configuration files), which are hidden in the wrappers.

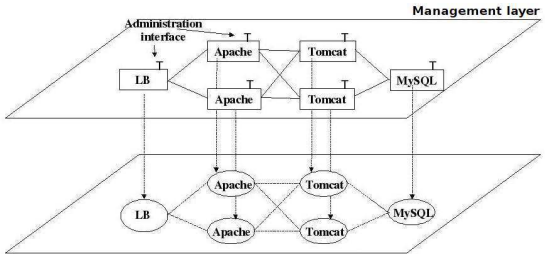

Fig. 2. Management layer for the J2EE application

Here, we distinguish two important roles:

- the role of the management and control interfaces is to provide a means for configuring components and bindings between components. It includes methods for navigating in the component-based management layer or modifying it to implement reconfigurations.
- the role of the wrappers is to reflect changes in the management layer onto the legacy layer. The implementation of a wrapper for a specific software may also have to navigate in the component management layer, to access key attributes of the components and generate legacy software configuration files[1].

2.3 Management Policy Specification

In a first prototype (Jade [2], a predecessor of TUNe), the implementation of management policies was directly relying on the interfaces of the Fractal component model:

- A wrapper was implemented as a Fractal component, developed in Java, which main role is to reflect management/control operations onto the legacy software. For instance, if we consider the wrapper of the Apache software, the assignment of the port attribute of the wrapper is reflected in the *httpd.conf* file in which the port attribute is defined. Similarly, setting up a binding between an Apache wrapper and a Tomcat wrapper is reflected at the legacy layer in the *worker.properties* file.
- the description of a software architecture to be deployed was described in a Fractal ADL file. This ADL file describes in an XML syntax the set of components (wrappers) to instanciate (which will in turn deploy the associated legacy software components), their bindings and their configuration attributes.
- reconfigurations were developed in Java, relying on Fractal APIs. These APIs allow invoking components' management interfaces or Fractal control interfaces for assigning components' attributes, adding/removing components and updating bindings between components.

Component-based autonomic computing has proved to be a very convenient approach. The experiments we conducted with this first prototype for managing J2EE infrastructures [2] (but also other distributed infrastructures such as Diet grid middleware [6]) validated this design choice.

But as our system was used by external users (external to our group), we rapidly observed that the interfaces of a component model are too low-level and difficult to use. In order to implement wrappers (to encapsulate existing software), to describe deployed architectures and to implement reconfiguration programs, the administrator of the environment has to learn (yet) another framework, the Fractal component model in our case. More precisely, our previous experiments showed us that:

[1] e.g. for configuring an Apache, we need to access attributes from both the Apache component and the Tomcat components it is bound with

– wrapping components is difficult to implement. The developper needs to have a good understanding of the component model we use (Fractal).
– architectures are not very easy to describe. ADLs are generally very verbose and still require a good understanding of the underlying component model. Moreover, if we consider large scale software infrastructure such as those deployed over a grid, describing an architecture composed of a thousand of servers requires an ADL description file of several thousands of lines.
– reconfiguration policies are difficult to implement as they have to be programmed using the management and control interfaces of the management layer. This also requires a strong expertise regarding the used component model.

3 A DSL based approach

The conclusions of the previous section led us to explore the introduction of higher level formalisms (or DSLs) for all the administration tasks (wrapping, configuration, deployment, reconfiguration). Our main motivation was to hide the details of the component model we rely on and to provide a more abstract and intuitive specification interface. In order to provide user-friendly, easy to learn and to use formalisms, the design of these languages is largely inspired by UML diagrams.

In the following sections, we present each of these DSLs and described their associated metamodels.

3.1 Configuration Description Language

Our previous experiments showed us that it is not easy to describe architectures with an ADL. We propose to describe an architecture with a graphical language, which is much more intuitive and easy to use than Fractal's ADL. Moreover, this DSL that we call the *Configuration Description Language* (CDL) is intensional. Intentional architecture definition means that each software can be instantiated in several replicas. A link between two software generates bindings between the replicas instanciated from these software. Each described software includes a set of configuration attributes which are specific to the software.

The corresponding metamodel is depicted in Figure 3(a). The main concept of this view is the *SoftwareElement* describing a particular type of software with its own configuration. Each *SoftwareElement* is described by a set of properties (*attributes*), with an initial value (*default*), which are used by the administrator to configure the legacy software that the *SoftwareElement* represents. Note that a *SoftwareElement* can be instantiated into several replicas. For a particular software, if we required several instances with different configuration properties, we obtain as much *SoftwareElement* as configurations, with different properties for each one. The architecture is intentionally described through the definition of bindings (*Link*), allowing to connect a *SoftwareElement* to another, and expressing a multiplicity (*lower* & *upper*) and a role (*name*). The multiplicity expresses

the range of instances of the target *SoftwareElement* for each one of the source *SoftwareElement*. The role allows navigation with a query language relying on OCL [7]. Bi-directional *bindings* is allowed by defining an *opposite bindings*.

In this metamodel, each concept defines several attributes which are predefined by TUNe and used for administration:

- **name** gives the name of the element (software, link or attribute). These names allow navigation in the architecture, which is usefull the definition of wrappers and reconfiguration policies (detailed later).
- **baseDirectory** gives the name of the repository where legacy software archives are available for installation.
- **legacyFile** gives the name of the archive which contains the legacy software binaries and configuration files.
- **lower & upper** gives respectively the minimal and the maximal cardinality of a link.

Figure 3(b) shows a CDL description for the J2EE example. In this architecture, the *Apache* web server can be instantiated in up to 3 replicas and each *Apache* instance can be linked with at least 1 instance of *Tomcat* and up to 4 instances.

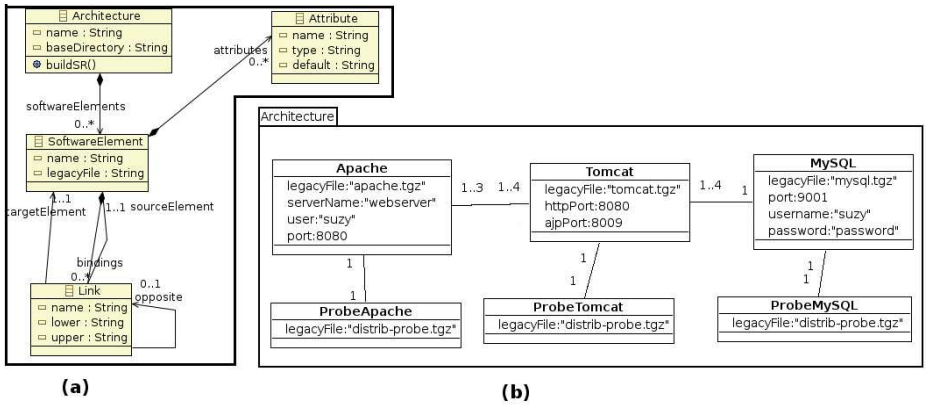

Fig. 3. (a) Configuration Description Language (b) Architecture schema

3.2 Deployment Description Language

In order to specify the deployment of an architecture in a computing environment composed of machines, we introduce a language called the *Deployment Descrition Language* (DDL). The main objective is to allow the definition of different deployment models according to the architecture model. As for the CDL, it takes advantage of the expressiveness of the graphical notation. It also allows

to define by intension or by extension, the real deployment of instances of each software component on clusters nodes. Attributes can be associated with clusters in order to describe the perperties of clusters' nodes. As for CDL, these attributes can be used to in the definition of wrappers and reconfiguration policies (detailed later).

The metamodel of the DDL is depicted in Figure 4(a). For each *SoftwareElement* a set of *Deployments* is defined, describing a real number of instances (*initial*) to be deployed on a *Cluster*. Each *Cluster* is described by a set of properties (attributes) and a *SoftwareElement* is deployed on a cluster with a defines deployment policy (*Policy*). A policy is implemented as a Java class.

Figure 4(b) shows a J2EE deployment shema where 2 *Apache*s and 1 *Tomcat* are deployed on *cluster1* using the deployment policy *policy.Cluster1* and 1 *Tomcat* and 1 *MySQL* are deployed on *Cluster2* with the deployment policy *policy.Cluster2*. The *SoftwareElement* concept in this language is the same in the CDL. For instance, the number of deployed instances must be compatible with the multiplicities described in the architecture. The clear separation of the deployment and architecture concerns allows to define several deployment policies for the same architecture.

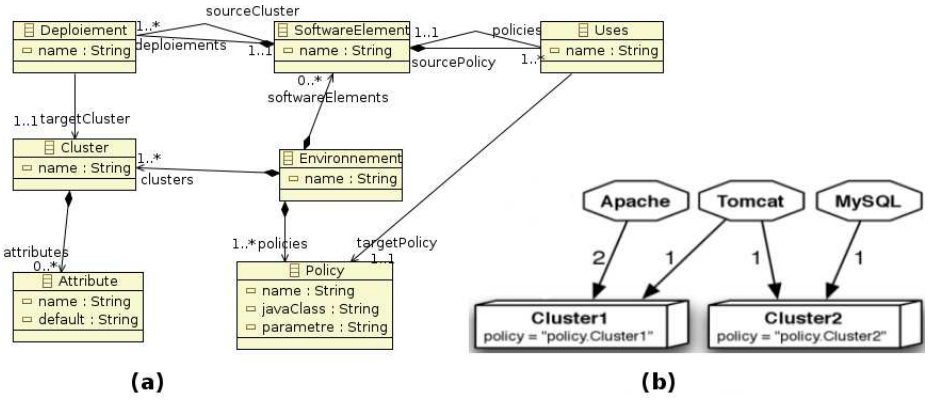

Fig. 4. (a) Deployment Description Language (b) Deployment policy

3.3 Wrapping Description Language

In order to simplify the definition of wrapper, we introduce a *Wrapping Description Language* (WDL). A WDL specification is interpreted by a generic wrapper Fractal component, which implements an equivalent wrapper. Therefore, an administrator doesn't have to program any implementation of Fractal component.

A WDL description defines a set of methods that can be invoked to (re)configure the wrapped software. The workflow of methods that have to be invoked in order

to (re)configure the overall software environment is defined thanks to a formalism introduced in Section 3.4. The WDL description provides the implementation of these methods, with for each method: (1) a reference to Java class which includes the actual Java implementation of the method, and (2) the parameters that should be passed to this Java method. These Java methods generally manipulate configuration files or run shell commands. We assume that most of the needs should be met with a finite set of generic Java methods implementations (that can be therefore reused).

The grammar of our WDL is given in Figure 5(a) and an example with the J2EE architecture is depicted in Figure 5(b).

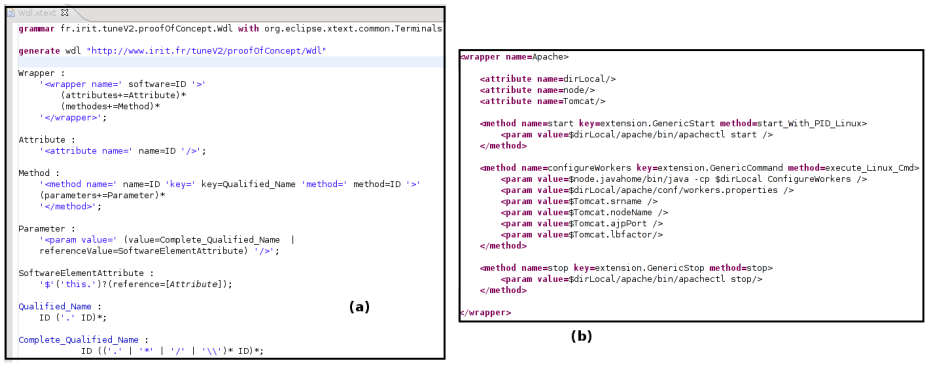

Fig. 5. (a) Wrapping Description Language (b) J2EE WDL

Methods' parameter may be *Attribute* values (defined in the architecture schema). It is sometimes necessary to navigate in the deployed component architecture in order to configure the software. For instance, in the J2EE architecture illustrated in Figure 3(b), an Apache may be bound to several Tomcats. At the legacy layer, the *worker.properties* configuration file of Apache must include the list of Tomcat nodes. Therefore, the *configureWorkers* method in the Apache wrapper must receive this list of nodes in order to configure the *worker.properties* file. *$Tomcat.nodeName* follows the links from Apache to the Tomcats it is bound with and obtains the *nodeName* attribute for each Tomcat.

This implies that this navigation must be consistent with the architectural schema described with the CDL. We have thus defined OCL constraints to verify that.

3.4 Reconfiguration Description Language

Regarding reconfiguration policies, we introduce a *Reconfiguration Description Language* (RDL) which allows to define workflows of operations that have to be performed for reconfiguring the managed environment. One of the main advantage of RDL, besides simplicity, is that it manipulates the entities described in

the deployment schema and reconfigurations can only produce a concrete architecture which comforms to the abstract schema, thus enforcing reconfiguration correctness.

Reconfigurations are triggered by events. An event can be generated by a specific monitoring component (e.g. probes in the architecture schema) or by a wrapped legacy software which already includes its own monitoring functions.

Operations defined in a workflow can assign an attribute or a set of attributes of components, or invoke a method or a set of methods of components. To designate the components on which the operations should be performed, the syntax of the operations allows navigation in the component architecture, similarly to the WDL. The grammar of this language is depicted in Figure 6(a).

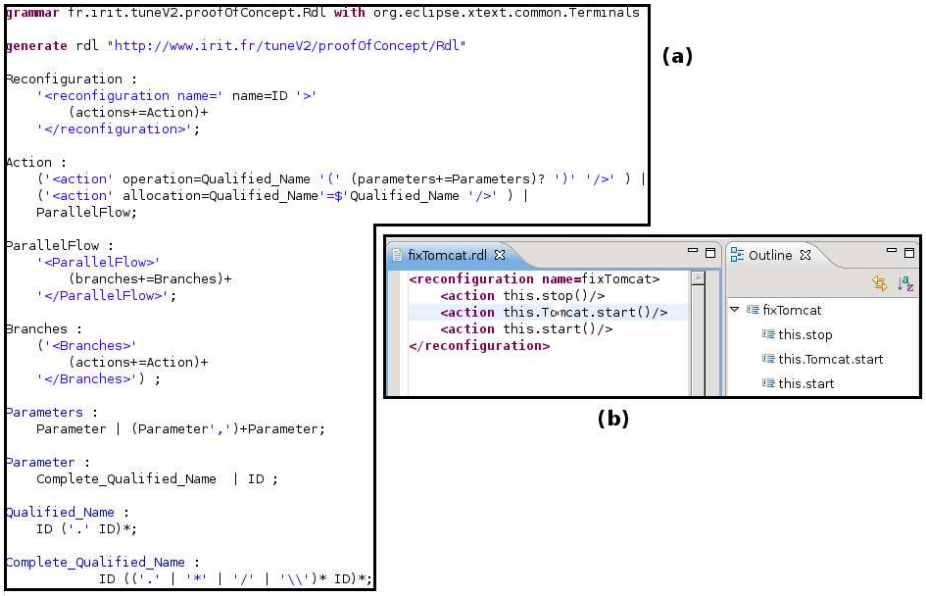

Fig. 6. (a) Reconfiguration Description Language (b) Tomcat repair

This language allows to apply *Action*s on the component architecture. *Action*s are either attribute assignments or method invocations and they can be executed at in parallel (*ParallelFlow*) or sequentially (*Branches*). Let's consider the example in Figure 6(b) which is the reaction to a Tomcat (software) failure. Event (*fixTomcat*) is generated by a *probeTomcat* component instance, therefore the *this* variable references this probe. Then:

– *this.stop* will invoke the *stop* method on the probing component (to prevent the generation of multiple events);

– *this.Tomcat.start* will invoke the *start* method on the Tomcat component instance linked with the probe. This is the actual repair of the faulting Tomcat server;
– *this.start* will restart the probe associated with the Tomcat.

Notice that operations are expressed using the elements defined in the architecture schema, but are applied on the actually deployed component architecture.

4 Implementation on Kermeta

The IDM community proposes a multitude of technologies/tools allowing to implement DSLs: metamodel edition, DSL editor generation, DSL tranformation and simulation/execution. Among these tools we can cite EMF [8], ATL [9] and Kermeta [10] environments. The work presented in this paper relies on the Kermeta framework. In this section, after a presentation of the Kermeta framework, we describe step by step the implementation of our approach.

Kermeta Kermeta is presented by their developpers as a metamodelisation kernel for addressing all metamodelisation problems. For our purpose, we rely on two basic features of Kermeta:

– Kermeta allows the graphical definition of a DSL's metamodel through a metamodel editor. A metamodel can also be defined in the EMF framework thanks to an eclipse plugin [11]. The EMF definition (saved in a .ecore file) is translatable to kermeta and vice versa. The advantage of EMF is that it allows the generation of specific editors (for editing models which conforms to the metamodel) associated with the defined DSL.
– Kermeta provides a metaprogramming language for the implementation of operations associated with DSL concepts. The Kermeta language is an object based language which looks like the Java language. Model parsing is the main objectif of this language. Besides this navigation, the kermeta language allows to strengthen (by constraints specification) the conformity between models and metamodels.

Besides these two main features, Kermeta also provides support for OCL constraints [7], execution of external programs (notably Java code), and it is integrated as a plugin in the eclipse framework.

In the following sections, we describe how we implemented our DSL with kermeta (version 1.2). This implementation is organised in several steps: DSL metamodel edition, DSL implementation, DSL editing tools and DSL execution.

DSL metamodel edition It is the first step in the implementation of a DSL. Except the RDL and WDL, the metamodel of all other DSLs are graphically edited:

– we edit the metamodels with the EMF framework (remind that it will be used to generate an editor for the DSL)
– we associate OCL constraints with this metamodel
– we translate the EMF metamodel in Kermeta

Figure 7 shows the translation of *CDL.ecore* (Figure 7(a)) in EMF format, to *CDL.kmt* (Figure 7(b)) in kermeta format. The graphical concept *SoftwareElement* and its attributes (Figure 7(a)) are translated to kermeta classes and data members (Figure 7(b)). Framed elements with the same color in Figure 7 are equivalent. The association between the *SoftwareElement* concept and the *Attribute* concept is translated into a reference attribute in the SoftwareElement class (green frame in Figure 7).

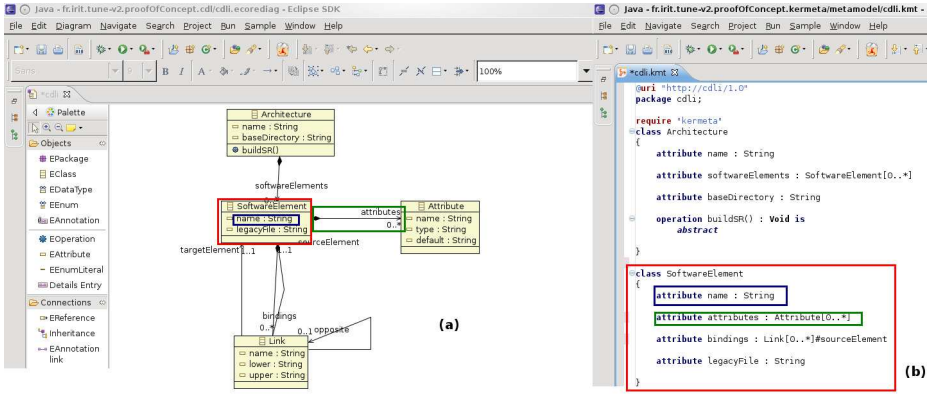

Fig. 7. CDL Definition

DSL Implementation After the previous structural definition of DSLs, kermeta allows to define the semantics of DSLs. It consists in defining the body of the methods associated with the various concepts. Let us take the example of the CDL to illustrate this implementation. In Figure 7(a), the *Architecture* concept defines a method *buildSR* (SR for System Representation, the implementation of the management layer described in Figure 2). The objective of this method is the construction of the system representation of the future administrated application: software component creation (*SoftwareElement* concept), attributes creation (*Attribute* concept) and links between software (*Link* concept). The implementation of the CDL is shown in Figure 8(a). The construction of the SR is performed only if the architecture to be administrated is in accordance with the set of defined OCL constraints (previous section). These OCL constraints were translated into Kermeta invariants which are checked in the *buildSR* method (green framed elements in the figure).

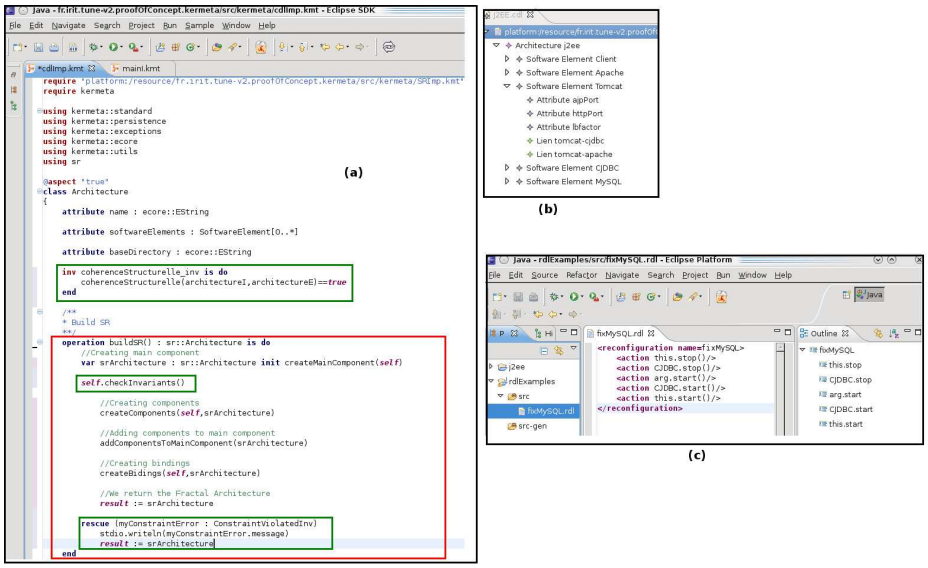

Fig. 8. (a) CDL Implementation. (b) CDL editor: J2EE architecture. (c) RDL editor: MySQL repair policy

Similarly, the semantics of the other DSLs are defined by following the same process.

DSL Editing Tools After the DSLs' definition (structure and semantic), we are interested in providing editing tools for administrators who will define policies with these DSLs.

The EMF framework allows to generate editors from a metamodel definition, but it can only generate textual and tree-like editors. Initially, we wanted to implement graphical editors for our DSLs (more precisely CDL and DDL). Unfortunately, there is a gap between expectations and reality in Model Driven Engineering. GMF is the framework advocated by Eclipse for building graphical editors and it turned out to be overly complex. In front of these difficulties, we are currently working on framework for graphical editor generation. So for the moment, the J2EE architecture presented in Section 2.1 can be edited in a generated tree-like editor. This editor supervises the edition of the architecture by insuring its conformance with the CDL definition.

Figure 8(b) shows this editor. A similar editor was generated for the DDL language. For textual languages (WDL and RDL), we generated dedicated editors with the xtext tool [12]. These editors are comparable to the *eclipse* java programming environment. They offer a syntactic recognition (in color), completion, inline constraints check, etc. Figure 8(c) shows an example of a reconfiguration policy written in the RDL editor for the repair of a MySQL server upon failure.

Once the various administration policies are defined with the DSLs and saved in files, the Kermeta virtual machine can be launched and given these files in order to execute the defined administration policy. In the next section, we describe this execution process.

DSL execution The execution of an administration policy defined with our DSLs is made at two levels. The first level is in the Kermeta environment. At this level, the Kermeta virtual machine executes the semantics of the DSLs (which are Kermeta programs) and applies them to the corresponding policies. In our case, the interpretations of policies are performed in the following order: CDL, WDL, DDl and RDL.

The second level of execution is in the Java virtual machine. Indeed, the accomplishment of management actions on the system representation and the legacy layer is realized through the Java API of the underlying component model (Fractal in our case). These calls are completely transparent for the administrator, who only interacts with the system through DSLs.

Figure 9 synthetizes the relationship between these two levels of execution.

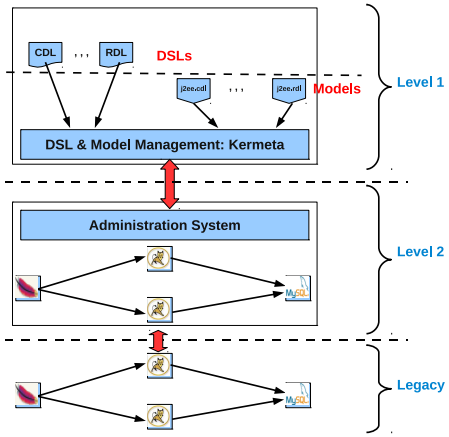

Fig. 9. DSL execution

We observed that the use of these two virtual machines slowed down the execution process. Indeed, the Kermeta virtual machine is also based on the Java virtual machine. In the following section, we look at other existing technologies used in our context.

5 Related Works

Autonomic computing is an appealing approach that aims at simplifying the hard task of system management, thus building self-healing, self-tuning, and self-configuring systems [13].

Management solutions for legacy systems are usually proposed as ad-hoc solutions that are tied to particular legacy system implementations (e.g. [14] for self-tuning cluster environments). This unfortunately reduces reusability and requires autonomic management procedures to be reimplemented each time a legacy system is taken into account in a particular context.

Relying on a component model for managing legacy software infrastructure has been investigated by several projects [1–3] and has proved to be a very convenient approach, but in most cases, the autonomic policies have to be programmed using the programming interface of the underlying component model (a framework for implementing wrappers, configuration APIs or deployment ADLs) which is too low level and still error prone.

Therefore, many projects explored model-driven approaches for designing autonomic management policies. Some of them proposed frameworks for modeling autonomic systems, e.g. a self-healing [15], a self-protecting [16], or a resource management system [17], the management system implementation being generated from the described management model. The modeling of such a system can still be quite complex and the integration within a legacy software organisation tricky. Some other projects proposed frameworks and runtimes for modeling the managed system and maintaining consistency between the managed system and its model at runtime [18, 19]. The main advantage is well defined representation of the managed system, on which management policies can be applied. The TUNe system falls into this category, even if our management layer relies on the Fractal component model.

TUNe relies on DSL for the specification of a software architecture, its deployment and reconfiguration. These languages ensure that only consistent system states can result from deployment and reconfiguration. Some projects considered interactions between policies, mainly in order to deal with conflicts [20, 21]. We are currently working on a DSL which should allow such coordinating between reconfiguration policies.

Concerning policy consistency verification, the most important task is done by generated editors. Many solutions were proposed in this domain. The EMF framework can generate an editor from a DSL, but it is limited to a tree-like editor, which does not enough expressiveness for models edition. Frameworks like GMF [22] and Epsilon EuGenia [23] suggest generating graphical editors. However these frameworks are difficult to use as the implementation of an editor for a DSL still requires a significant development effort. The Obeo Design [24] framework is targetting to address this issue, but at the time of writting, we were not able to experiment with their tool.

6 Conclusion and Perspectives

We designed and implemented an autonomic management system called TUNe which relies on a component model in order to manage a software environment as a component architecture. However, we observed that the interfaces of a component model are too low-level and difficult to use. For this reason, we designed higher abstraction level languages (DSLs) for modeling administration policies. In order to implement these DSLs, we experimented with techniques from the Model Driven Engineering community. We defined the metamodels of the DSLs and implemented their semantics with the Kermeta framework. From the metamodels of the DSLs, dedicated editors can be generated, but these editors are very limited, in particular it is difficult to build editors for graphical languages.

The main continuation of this work is related to this later point. We are currently investigating the implementation of a framework which should allow describing at the level of the metamodel of a graphical DSL the graphical representation associated with this language. This framework should allow generation of graphical DSL specific editors without any development.

References

1. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self adaptation with reusable Infrastructure. In: IEEE Computer, 37(10). (2004)
2. Hagimont, D., Bouchenak, S., Palma, N.D., Taton, C.: Autonomic Management of Clustered Applications. In: IEEE International Conference on Cluster Computing. (2006)
3. Oriezy, P., Gorlick, M., Taylor, R., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., A.Wolf: An Architecture-Based Approach to Self-Adaptive Software. In: IEEE Intelligent Systems 14(3). (1999)
4. Microsystems, S.: Java 2 Platform Enterprise Edition (J2EE). http://java.sun.com/j2ee/ ( )
5. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: The Fractal Component Model and its Support in Java. In: Software - Practice and Experience, special issue on *Experiences with Auto-adaptive and Reconfigurable Systems*. (2006)
6. Chebaro, O., Broto, L., Bahsoun, J.P., Hagimont, D.: Self-tune-ing of a j2ee clustered application. In: Proceedings of the 2009 Sixth IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems, Washington, DC, USA, IEEE Computer Society (2009) 23–31
7. Object Management Group: UML Object Constraint Language (OCL) 2.0. (2005)
8. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., Gronback, R.C., Milinkovich, M.: EMF: Eclipse Modelign Framework; 2nd ed. The eclipse series. Addison-Wesley, Upper Saddle River, NJ (2009)
9. Bezivin, J., Dupe, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the atl model transformation language: Transforming xslt into xquery. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)
10. Drey, Z., Faucher, C., Fleurey, F., Vojtisek, D.: Kermeta language reference manual. (2006)

11. Eclipse: Eclipse. http://www.eclipse.org/ (visiting at 2010)
12. Open Architecture Ware: xtext. http://www.openarchitectureware.org/ (2007)
13. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. In: IEEE Computer Magazine, 36(1). (2003)
14. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P.: Dynamic Provisiong of Multi-Tier Internet Applications. In: 2nd International Conference on Autonomic Computing. (2005)
15. Jiang, M., Zhang, J., Raymer, D., Strassner, J.: A Modeling Framework for Self-Healing Software Systems. In: Workshop "Models@run.time" at the 10th International Conference on model Driven Engineering Languages and Systems. (2007)
16. Pena, J., Hinchey, M.G., Sterritt, R., Ruiz-Cortes, A., Resinas, M.: A model-driven architecture approach for modeling, specifying and deploying policies in autonomous and autonomic systems. In: 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing, IEEE Computer Society (2006) 19–30
17. Barrett, K., Davy, S., Strassner, J., Jennings, B., van der Meer, S., Donnelly, W.: A model based approach for policy tool generation and policy analysis. In: Proc. 1st IEEE Global Information Infrastructure Symposium, IEEE (2007) 99–105
18. Sriplakich, P., Waignier, G., Le Meur, A.F.: Enabling Dynamic Co-Evolution of Models and Runtime Applications. In: 1st IEEE International Workshop on Model-Driven Development of Autonomic Systems, IEEE Computer Society (2008)
19. Rohr, M., Boskovic, M., Giesecke, S., Hasselbring, W.: Model-driven Development of Self-managing Software Systems. In: Workshop "Models@run.time" at the 9th International Conference on model Driven Engineering Languages and Systems (MoDELS). (2006)
20. Agrawal, D., Lee, K.W., Lobo, J.: Policy-based management of networked computing systems. Communications Magazine, IEEE **43**(10) (2005) 69–75
21. Davy, S., Barrett, K., Serrano, M., Strassner, J., Jennings, B., van der Meer, S.: Policy Interactions and Management of Traffic Engineering Services Based on Ontologies. Network Operations and Management Symposium (2007) 95–105
22. GMF: Graphical Modeling Framework. http://www.eclipse.org/gmf/ ( )
23. project, E.: EuGenia. http://www.eclipse.org/gmt/epsilon/doc/eugenia/ ( )
24. Company, O.M.D.: Obeo Designer white paper. http://www.obeo.fr/resources/FicheProduit-OD.pdf ( )